fork，vfork，clone 与 pthread_create 的区别

进程是一个指令执行流及其执行环境，其执行环境是一个系统资源的集合，这些资源在 Linux 中被抽象成各种数据对象：进程控制块、虚存空间、文件系统，文件 I/O、信号处理函数。所以创建一个进程的过程就是这些数据对象的创建过程。

在调用系统调用 fork 创建一个进程时，子进程只是完全复制父进程的资源，这样得到的子进程独立于父进程，具有良好的并发性，但是二者之间的通讯需要通过专门的通讯机制，如：pipe，fifo，System V，IPC 机制等，另外通过 fork 创建子进程系统开销很大，需要将上面描述的每种资源都复制一个副本。这样看来，fork 是一个开销十分大的系统调用，这些开销并不是所有的情况下都是必须的，比如某进程 fork 出一个子进程后，其子进程仅仅是为了调用 exec 执行另一个执行文件，那么在 fork 过程中对于虚存空间的复制将是一个多余的过程（由于 Linux 中是采取了 copy-on-write 技术，所以这一步骤的所做的工作只是虚存管理部分的复制以及页表的创建，而并没有包括物理页面的拷贝）。另外，有时一个进程中具有几个独立的计算单元，可以在相同的地址空间上基本无冲突进行运算，但是为了把这些计算单元分配到不同的处理器上，需要创建几个子进程，然后各个子进程分别计算最后通过一定的进程间通讯和同步机制把计算结果汇总，这样做往往有许多格外的开销，而且这种开销有时足以抵消并行计算带来的好处。

这说明了把计算单元抽象到进程上是不充分的，这也就是许多系统中都引入了线程的概念的原因。在讲述线程前首先介绍以下 vfork 系统调用，vfork 系统调用不同于 fork，用 vfork 创建的子进程共享地址空间，也就是说子进程完全运行在父进程的地址空间上，子进程对虚拟地址空间任何数据的修改同样为父进程所见。但是用 vfork 创建子进程后，父进程会被阻塞直到子进程调用 exec 或 exit。这样的好处是在子进程被创建后仅仅是为了调用 exec 执行另一个程序时，因为它就不会对父进程的地址空间有任何引用，所以对地址空间的复制是多余的，通过 vfork 可以减少不必要的开销。在 Linux 中， fork 和 vfork 都是调用同一个核心函数 do_fork(unsigned long clone_flag, unsigned long usp, struct pt_regs)其中 clone_flag 包括 CLONE_VM, CLONE_FS, CLONE_FILES, CLONE_SIGHAND, CLONE_PID ，CLONE_VFORK 等标志位，任何一位被置 1 了则表明创建的子进程和父进程共享该位对应的资源。所以在 vfork 的实现中，cloneflags = CLONE_VFORK | CLONE_VM | SIGCHLD，这表示子进程和父进程共享地址空间，同时 do_fork 会检查 CLONE_VFORK，如果该位被

置 1 了，子进程会把父进程的地址空间锁住，直到子进程退出或执行 exec 时才释放该锁。

在讲述 clone 系统调用前先简单介绍线程的一些概念。

线程是在进程的基础上进一步的抽象，也就是说一个进程分为两个部分：线程集合和资源集合。线程是进程中的一个动态对象，它应该是一组独立的指令流，进程中的所有线程将共享进程里的资源。但是线程应该有自己的私有对象：比如程序计数器、堆栈和寄存器上下文。线程分为三种类型：内核线程、轻量级进程和用户线程。

内核线程：

它的创建和撤消是由内核的内部需求来决定的，用来负责执行一个指定的函数，一个内核线程不需要和一个用户进程联系起来。它共享内核的正文段和全局数据，具有自己的内核堆栈。它能够单独的被调度并且使用标准的内核同步机制，可以被单独的分配到一个处理器上运行。内核线程的调度由于不需要经过态的转换并进行地址空间的重新映射，因此在内核线程间做上下文切换比在进程间做上下文切换快得多。

轻量级进程：

轻量级进程是核心支持的用户线程，它在一个单独的进程中提供多线程控制。这些轻量级进程被单独的调度，可以在多个处理器上运行，每一个轻量级进程都被绑定在一个内核线程上，而且在它的生命周期这种绑定都是有效的。轻量级进程被独立调度并且共享地址空间和进程中的其它资源，但是每个 LWP 都应该有自己的程序计数器、寄存器集合、核心栈和用户栈。

用户线程：

用户线程是通过线程库实现的。它们可以在没有内核参与下创建、释放和管理。线程库提供了同步和调度的方法。这样进程可以使用大量的线程而不消耗内核资源，而且省去大量的系统开销。用户线程的实现是可能的，因为用户线程的上下文可以在没有内核干预的情况下保存和恢复。每个用户线程都可以有自己的用户堆栈，一块用来保存用户级寄存器上下文以及如信号屏蔽等状态信息的内存区。线程库库通过保存当前线程的堆栈和寄存器内容载入新调度线程的那些内容来实现用户线程之间的调度和上下文切换。内核仍然负责进程的切换，因为只有内核具有修改内存管理寄存器的权力。用户线程不是真正的调度实体，内核对它们一无所知，而只是调度用户线程下的进程或者轻量级进程，这些进程再通过线程库函数来调度它们的线程。当一个进程被抢占时，它的所有用户线程都被抢占，当一个用户线程被阻塞时，它会阻塞下面的轻量级进程，如果进程只有一个轻量级进程，则它的所有用户线程都会被阻塞。

明确了这些概念后，来讲述 Linux 的线程和 clone 系统调用。

在许多实现了多线程的操作系统中（如：Solaris，Digital Unix 等），线程和进程通过两种数据结构来抽象表示：进程表项和线程表项，一个进程表项可以指向若干个线程表项，调度器在进程的时间片内再调度线程。但是在 Linux 中没有做这种区分，而是统一使用 task_struct 来管理所有进程/线程，只是线程与线程之间的资源是共享的，这些资源可以是前面提到过的：虚存、文件系统、文件 I/O 以及信号处理函数甚至 PID 中的几种。也就是说 Linux 中，每个线程都有一个 task_struct，所以线程和进程可以使用同一调度器调度。其实 Linux 核心中，轻量级进程和进程没有质上的差别，因为 Linux 中进程的概念已经被抽象成了计算状态加资源的集合，这些资源在进程间可以共享。如果一个 task 独占所有的资源，则是一个 HWP，如果一个 task 和其它 task 共享部分资源，则是 LWP。clone 系统调用就是一个创建轻量级进程的系统调用：

```
int clone(int (*fn)(void * arg), void *stack, int flags, void * arg);
```

其中 fn 是轻量级进程所执行的过程，stack 是轻量级进程所使用的堆栈，flags 可以是前面提到的 CLONE_VM, CLONE_FS, CLONE_FILES, CLONE_SIGHAND,CLONE_PID 的组合。

Clone 和 fork，vfork 在实现时都是调用核心函数 do_fork。do_fork(unsigned long clone_flag, unsigned long usp, struct pt_regs)，和 fork、vfork 不同的是，fork 时 clone_flag = SIGCHLD；vfork 时 clone_flag = CLONE_VM | CLONE_VFORK | SIGCHLD；而在 clone 中，clone_flag 由用户给出。下面给出一个使用 clone 的例子。

```
Void * func(int arg)
{
. . . . . .
}
int main()
{
int clone_flag, arg;
. . . . . .
clone_flag = CLONE_VM | CLONE_SIGHAND | CLONE_FS |
CLONE_FILES;
stack = (char *)malloc(STACK_FRAME);
stack += STACK_FRAME;
```

```
retval = clone((void *)func, stack, clone_flag, arg);
. . . . . .
}
```

看起来 clone 的用法和 pthread_create 有些相似，两者的最根本的差别在于 clone 是创建一个 LWP，对核心是可见的，由核心调度，而 pthread_create 通常只是创建一个用户线程，对核心是不可见的，由线程库调度。linux 的 pthread_create 最终调用 clone，pthread_create 调用 clone，并把开辟一个 stack 作为参数 thread 建立，同步销毁等由线程库负责。